Implementation of a least-squares preconditioner for stabilized discretizations of the Navier-Stokes equations

Simon Funke

November 28, 2008

Contents

Abstract

This work describes the implementation of a least squares commutator preconditioner introduced by Elman at al. [*SIAM J. Sci Comput.*, 30(1): pp. 290-311, 2007] in Trilinos. This preconditioner is applicable to problems arising from stabilized low-order discretizations of the incompressible Navier Stokes equation. Due to the purely algebraic design no additional data is necessary for using the preconditioner. The software is written as a part of the preconditioning package within Trilinos which allows user-friendly use of the new algorithm.

1 Introduction

Consider the Navier-Stokes equation

$$\alpha\vec{u}_t - \nu\Delta\vec{u} + (\vec{u}\cdot\nabla)\vec{u} + \nabla p = \vec{f} \tag{1.1}$$

$$-\text{div } \vec{u} = 0 \tag{1.2}$$

where $\nu > 0$ is the kinematic viscosity on $\Omega \subset \mathbb{R}^n$ and $n = 2, 3$. The source function $\vec{f}$ is given on the domain $\Omega \subset \mathbb{R}^n$. The unknown functions $\vec{u} : \Omega \to \mathbb{R}^n$ and $p : \Omega \to \mathbb{R}$ represent the velocity and pressure of the fluid, respectively. $\alpha = 0$ corresponds to the stationary problem and $\alpha = 1$ to an unsteady flow.

The stationary Navier-Stokes equation is discretized with the finite element method. In the case of unsteady flow, in addition the horizontal (Rohte) or vertical line method is applied to discretize the time. In both cases, this leads to a nonlinear algebraic system. The nonlinear term arises from the convection term $(\vec{u}\cdot\nabla)\vec{u}$ in (1.1). By solving this nonlinear system with Newton or Picard iteration, a sequence of linear equations

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \tag{1.3}$$

has to be solved in each time step.

Unfortunately there are discretizations where (1.3) has no unique solution. These arise from unstable elements in the finite element method and include low order and equal order elements like $\mathbf{Q}_1 - \mathbf{P}_0$, $\mathbf{Q}_1 - \mathbf{Q}_1$ and $\mathbf{P}_1 - \mathbf{P}_1$. Since these elements are interesting in a numerical point of view, (1.3) is stabilized by replacing the $(2, 2)$ zero block with a stabilization matrix C:

$$\begin{bmatrix} F & B^T \\ B & -C \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \tag{1.4}$$

Linear systems like (1.3) or (1.4) can be high-dimensional and have to be solved several times in each time step. Therefore, the development for fast solvers is essential. Although it is possible to use direct solver for sparse datastructure, this paper concentrates on iterative solving methods. F is non-symmetric which excludes the use of the CG method. Instead GMRES, a Krylov subspace method is used together with a suitable preconditioner. Preconditioning is extremely important to guaranty fast convergence of GMRES, ideally independently bounded of the problem parameters.

Our approach of interest is preconditioning with the Schur complement. Consider the block LU-decomposition of the matrix in (1.4).

$$\begin{bmatrix} F & B^T \\ B & -C \end{bmatrix} = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \underbrace{\begin{bmatrix} F & B^T \\ 0 & -(BF^{-1}B^T + C) \end{bmatrix}}_{:=M} \tag{1.5}$$

If the second operator on the right hand side is viewed as a (right) preconditioner M, all eigenvalues of the preconditioned system would be identically one. In this case, the $GMRES$ algorithm applied to the preconditioned system would stop after two iterations [12], independent of the mesh size and the viscosity parameter.

The term $(BF^{-1}B^T + C) =: S$ in (1.5) is known as Schur complement. For unstabilized discretizations $C = 0$ and S becomes $BF^{-1}B^T$. Using M as a preconditioner implies that a linear problem with M has to be solved. Since the Schur complement is a full matrix it is not feasible to use S in M. Instead, it is replaced by an approximation $M_S \approx S$ which should be inexpensive to solve and in addition should be a good representation of S.

Finally, the preconditioning operator has the form

$$M = \begin{bmatrix} F & B^T \\ 0 & -M_S \end{bmatrix} \tag{1.6}$$

whose inverse can be decomposed to

$$M^{-1} = \begin{bmatrix} F^{-1} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & -B^T \\ 0 & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -M_S^{-1} \end{bmatrix} \tag{1.7}$$

The next chapters will present two strategies to compute M_S. Since the Schur complement is different for stable and unstable elements, it is not surprising that there are different approaches for M_S, too. The following chapter is concerned with stable discretization while section 3 will extend this preconditioner for the case of unstable discretizations, that is if $C \neq 0$. A more complete overview of different preconditioners can be found in [8].

2 LSC preconditioner

Introduced by Elman, Howle, Shadid, Shuttleworth and Tuminaro [9], the least squares commutator preconditioner is designed for stable discretizations ($C = 0$).

To motivate the LSC method, consider the linear problem (1.3) arising from using Picard iteration for the nonlinear system. F contains a scaled discrete version of the convection-diffusion operator $\mathcal{L} = -\nu\Delta + \vec{u} \cdot \nabla$ on the velocity space, where $\vec{u}$ is the velocity vector of the last Picard iteration. Let

us assume that there is an analogue operator $\mathcal{L}_p = (-\nu\Delta + \vec{u}\cdot\nabla)_p$ on the pressure space. The fact that this differential operator is not well-defined on the pressure space $L_2(\Omega)$ (and in turn $\mathcal{L}_p$) will be ignored.

The term of interest is the commutator of these operators with the gradient operator: $\mathcal{E} := \mathcal{L}\nabla - \nabla\mathcal{L}_p$. Motivated by the fact that $\mathcal{E} = 0$ if $\vec{u}$ is constant, the derivation of the LSC continues with the assumption that $\mathcal{E}$ is small in some sense.

Let Q and Q_p be the mass matrices on the velocity space and on the pressure space, respectively. Applying the finite element method to the gradient operator on the pressure space leads to its matrix representation $Q^{-1}B^T$. If $\mathcal{L}$ and $\mathcal{L}_p$ are discretized analogous, the matrix representation of the commutator $\mathcal{E}$ can be formed:

$$\mathcal{E}_h = (Q^{-1}F)(Q^{-1}B^T) - (Q^{-1}B^T)(Q_p^{-1}F_p) \tag{2.1}$$

Multiplying with $BF^{-1}Q$ from the left and with $F_p^{-1}Q_p$ from the right together with the assumption $\mathcal{E}_h \approx 0$ isolates the Schur complement:

$$BF^{-1}B^T \approx BQ^{-1}B^TF_p^{-1}Q_p \tag{2.2}$$

This derivation leads to a first approximation of S. Indeed, (2.2) is still not applicable since it contains the unknown operator F_p. To bypass this problem we will now construct an approximation of F_p in respect of minimizing $\mathcal{E}_h$. This minimizing process of $\mathcal{E}_h$ is done for each column of F_p individually. That is, for each j find $[F_p]_j$ which minimizes the term

$$||[Q^{-1}FQ^{-1}B^T]_j - Q^{-1}B^TQ_p^{-1}[F_p]_j||_Q \tag{2.3}$$

where $||\cdot||_Q$ is the induced norm of the discretized L_2 scalar product on the discrete velocity space $(u,v)_Q = (Qu,v)$.

The appropriate normal equation reads as:

$$Q_p^{-1}BQ^{-1}B^TQ_p^{-1}[F_p]_j = [Q_p^{-1}BQ^{-1}FQ^{-1}B^T]_j \tag{2.4}$$

Therefore, F_p can be be written as:

$$F_p = Q_p(BQ^{-1}B^T)^{-1}(BQ^{-1}FQ^{-1}B^T) \tag{2.5}$$

Substitution of F_p in (2.2) leads to following approximation of the Schur complement:

$$BF^{-1}B^T \approx (BQ^{-1}B^T)(BQ^{-1}FQ^{-1}B^T)^{-1}(BQ^{-1}B^T) \tag{2.6}$$

A small modification is needed to finally obtain the LSC preconditioner. For many discretizations the inverse of the velocity mass matrix Q^{-1} is dense and

is not practical to form. Instead, mass lumping is used where Q is replaced by a simple matrix $\hat{Q}$ (a common choice is the restriction to the diagonal $\hat{Q} = diag(Q)$). Altogether, the LSC approximates the inverse of the Schur complement by

$$M_S^{-1} = (B\hat{Q}^{-1}B^T)^{-1}(B\hat{Q}^{-1}F\hat{Q}^{-1}B^T)(B\hat{Q}^{-1}B^T)^{-1} \tag{2.7}$$

Therefore, solving a linear equation with M_S requires two discrete Laplacian operators and one multiplication with $B\hat{Q}^{-1}F\hat{Q}^{-1}B^T$. There is another approach called PCD preconditioner, which avoids the solution of one Laplacian operator by expecting the matrix F_p in (2.2) as an input matrix. However, the principal advantage of LSC is the fully automated construction from the block matrices F, B^T, B and the mass matrix only.

In Table 1 a comparison of the key properties for the most common block preconditioner is shown.

Properties	PCD	LSC	SLSC
Applies to stabilized approximation	yes	no	yes
Handles non-uniform meshes	yes	yes	yes
Offers a fully automated construction	no	yes	yes

Table 1: Comparison of some key properties of three Schur complement preconditioners: pressure convection-diffusion (PCD), least squares commutator (LSC) and least squares commutator for stabilized discretization (SLSC)

3 LSC preconditioner for stabilized discretizations

Low-order elements like $\mathbf{P}_1 - \mathbf{P}_0$ and equal-order elements like $\mathbf{P}_1 - \mathbf{P}_1$ for triangular finite elements (respectively $\mathbf{Q}_1 - \mathbf{P}_0$ and $\mathbf{Q}_1 - \mathbf{Q}_1$ for rectangular finite elements) are interesting in a practical point of view and can be found in many software codes. Unfortunately these elements do not satisfy the inf-sup condition which implies the need for stabilization. As a consequence the LSC preconditioner is not applicable anymore. Elman et al. [10] published an extension of the LSC preconditioner (SLSC) for stabilized low order finite element approximations methods. The following section shows how to modify the LSC preconditioner in the case of unstable elements. Then a fully algebraic version is presented which is also the algorithm implemented in this work.

3.1 Element based LSC

The LSC preconditioner is a good choice for stable discretizations. However, it can not applied to stabilized problems. The crucial problem are the terms $BQ^{-1}B^T$ and $BQ^{-1}FQ^{-1}B^T$ whose composition form the approximation M_S^{-1}, see (2.6). While for stable discretization these terms represent their continues analogue, this property gets lost with unstable discretizations. Therefore, the terms need to be stabilized in such a way that they keep their representation for unstable discretizations, too.

We first address the term $BQ^{-1}B^T$. The associated continuous analogue is the *potential flow* problem: Find eigenvalues and eigenvectors satisfying

$$\begin{aligned} \vec{u} + \ \nabla p &= \lambda \vec{u} \\ -\text{div}\ \vec{u} &= -\lambda \Delta p \end{aligned} \tag{3.1}$$

with $\vec{u} = 0$ on $\partial\Omega$. Applying the finite element method leads to:

$$\begin{bmatrix} Q & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \lambda \begin{bmatrix} Q & 0 \\ 0 & A_p \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} \tag{3.2}$$

A_p stands for the discrete Laplacian operator defined on the pressure space. Eliminating u in (3.2) results in a generalized eigenvalue problem involving the appropriate Schur complement:

$$BQ^{-1}B^T p = \sigma A_p p \tag{3.3}$$

with $\sigma := \lambda(\lambda - 1)$. This is also related to an alternative inf-sup condition (see [11], pp. 272-273). For stable discretizations the eigenvalues of (3.3) satisfy:

$$0 = \sigma_1 < \sigma_* \leq \sigma_2 \leq \sigma_3 \leq ... \leq \sigma_{n_p} \leq \sigma^* < \infty \tag{3.4}$$

where $\sigma_*, \sigma^* > 0$ are mesh-size independent constants and n_p is the dimension of the discrete pressure space. Note that $\sigma_1 = 0$ because it is $BQ^{-1}B^T\mathbf{1} = 0$ when $\mathbf{1}$ is the constant vector. In other words, $BQ^{-1}B^T$ is spectrally equivalent to a discrete Laplacian on the pressure space.

This will not be true for unstable discretizations any more. In fact, σ_* and σ^* condition (3.4) will then depend on the mesh-size. In order to fix this, a stabilization matrix C_1 is defined which replaces the zero block in (3.2) on the left hand side:

$$\begin{bmatrix} Q & B^T \\ B & -C_1 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \lambda \begin{bmatrix} Q & 0 \\ 0 & A_p \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} \tag{3.5}$$

For stabilized discretizations, a suitable generalization of the inf-sup constant σ_* is derived by replacing the Schur complement in (3.3) by its stabilized version. This results to following generalized eigenvalue problem:

$$(BQ^{-1}B^T + C_1)p = \delta A_p p \tag{3.6}$$

Then, C_1 is to be designed in such a way, that the eigenvalues of this generalized eigenvalue problem satisfy

$$0 = \delta_1 < \delta_* \leq \delta_2 \leq \delta_3 \leq ... \leq \delta_{n_p} \leq \delta^* < \infty \tag{3.7}$$

where $\delta_*, \delta^* > 0$ are mesh-size independent constants.

The second critical term, $BQ^{-1}FQ^{-1}B^T$ is stabilized in the same way. A stabilization matrix C_2 is defined and is added to the term. Replacing the terms of the LSC preconditioner by their stabilized analogue leads to a first version of the stabilized LSC:

$$M_S^{-1} = (BQ^{-1}B^T + C_1)^{-1}(BQ^{-1}FQ^{-1}B^T + C_2)(BQ^{-1}B^T + C_1)^{-1} \tag{3.8}$$

Examples for the construction of C_1 and C_2 can be found in [10]. Though, in all cases C_1 and C_2 are based on the stabilization method for unstable finite elements (i.e. the method to create C in (1.4)).

3.2 Fully algebraic LSC (SLSC)

As mentioned above, the element based LSC uses information from the stabilization method to create the preconditioner. Particularly, the construction needs access to the assembling process of the stabilization method. In many cases it is either not possible or not practical to access to information of the assembling code. Instead, what is needed is an extension of (3.8) to a fully algebraic approach. That is the preconditioner should be constructed by only using the block matrices F, B^T, B and C of (1.4) and the mass lumping matrix $\hat{Q}$.

The following algorithm presents the fully algebraic version introduced by Elman et al. For a detailed derivation see [10].

Algorithm 1. *The inverse Schur complement for stabilized uniform meshes can be approximated by*[1]*:*

$$M_S^{-1} = M_{S_\gamma}^{-1} + M_{S_\alpha}^{-1} \tag{3.9}$$

[1]Note: There is a sign mistake in the definition of α in [10].

with

$$M_{S_\gamma}^{-1} = (BQ^{-1}B^T + \gamma C)^{-1}(BQ^{-1}FQ^{-1}B^T)(BQ^{-1}B^T + \gamma C)^{-1} \tag{3.10}$$

and

$$M_{S_\alpha}^{-1} = \alpha D^{-1} := \alpha(diag(Bdiag(F)^{-1}B^T + C))^{-1} \tag{3.11}$$

α and γ are defined as $\alpha = \frac{1}{\rho(Bdiag(F)^{-1}B^T D^{-1})}$ and $\gamma = \frac{\rho(F)}{3}$

To apply this algorithm for non-uniform meshes, a small modification is necessary. In (3.10), the term γC represents the stabilization matrix C_1 in (3.8). Due to uniformity of the mesh in algorithm 1, it was possible to scale C with a appropriate number to achieve a reasonable choice of C_1. In the non-uniform case, C must be scaled with respect to the local element area. This is done by multiplication of C with a diagonal matrix D_r whose diagonal elements are defined as $(D_r)_{ii} = \frac{(BQ^{-1}B^T)_{ii}}{C_{ii}}$. With $\tilde{D}_r = \frac{D_r}{||D_r||_\infty}$ the SLSC for non-uniform meshes is:

$$M_S^{-1} = (BQ^{-1}B^T + \gamma \tilde{D}_r^{\frac{1}{2}} C \tilde{D}_r^{\frac{1}{2}})^{-1}(BQ^{-1}FQ^{-1}B^T)(BQ^{-1}B^T + \gamma \tilde{D}_r^{\frac{1}{2}} C \tilde{D}_r^{\frac{1}{2}})^{-1} + M_{S_\alpha}^{-1} \tag{3.12}$$

On the other side, if (3.12) is applied to a uniform mesh, D_r will be a scaled identity matrix and therefore $\tilde{D}_r = id$. That is, (3.12) is equivalent to algorithm 1 in case of uniform meshes.

As in the derivation of LSC, mass lumping is used to achieve a feasible algorithm. Therefore, in the calculation of M_S^{-1} instead of the mass matrix Q an approximation $\hat{Q}$ is used (e.g. $\hat{Q} = diag(Q)$).

4 Implementation of the SLSC preconditioner

4.1 Software environment

The main part of this work is the implementation of the SLSC preconditioner in Trilinos. Trilinos is a software package of parallel solver algorithms and libraries for the solution of large-scale, complex multi-physics engineering and scientific applications. The implementation of SLSC depends on several packages within Trilinos.

Epetra Epetra provides basic classes for constructing and manipulating matrix and vector objects. Parallelism is also included, which allows fast computation on distributed systems. Epetra data types are supported as base data types by all following packages.

Thyra This package provides a common abstract interface to vectors, vector spaces and linear operators used by many solvers in Trilinos. The concrete implementation of the data types is usually Epetra. Both Thyra and Epetra data types will be used in the SLSC preconditioner.

Anasazi Anasazi is a framework for large-scale eigenvalue algorithms. This package is used for eigenvalue problems which occur during the computation of the SLSC preconditioner.

Stratimikos The package Stratimikos contains a unified set of Thyra-based wrappers to linear solver and preconditioner capabilities in Trilinos. Stratimikos is used to solve the subproblems arising in the SLSC preconditioner. With Stratimikos it's an easy task to test a wide variety of different solvers for these subproblems. In particular, in this paper the AztecOO and Amesos solver packages will be used with Stratimikos.

AztecOO AztecOO is a package comprising a set of parallel iterative solvers and preconditioners. It includes Krylov subspace methods like GMRES and CG.

Amesos Amesos is a package comprising a set of direct solvers for sparse linear problems. It supports third party libraries like SuperLU [6] and DSCPACK [2], two fast direct solvers used for the numerical results in section 5.

Meros Meros is a preconditioning package within Trilinos. Meros provides scalable block preconditioning for problems that couple simultaneous solution variables such as Navier-Stokes problems. The SLSC preconditioner presented here is implemented in Meros.

4.2 Description of the SLSC preconditioner class

As mentioned above, the SLSC preconditioner extends the Meros package. The preconditioner consists of the two classes *SLSCOperatorSource* and *SLSCPreconditionerFactory* which implement the virtual template classes *Thyra::LinearOpSourceBase* and *Thyra::PreconditionerFactoryBase*, respectively. This design ensures that the SLSC preconditioner can be handled in the same manner as any other preconditioner which implements the Thyra interface. In particular existing code can be easily adapted to use the SLSC preconditioner.

A brief introduction to the main functions is given here:

```
class SLSCOperatorSource : virtual public LinearOpSourceBase<double>
    {
    public:
      /** \brief Construct with epetra operators
       */
      SLSCOperatorSource(Epetra_CrsMatrix* S00,
                         Epetra_CrsMatrix* S01,
                         Epetra_CrsMatrix* S10,
                         Epetra_CrsMatrix* S11,
                         Epetra_Vector* Qu);
 [...]
}
```

```
 class SLSCPreconditionerFactory    : public Thyra::PreconditionerFactoryBase<double>
    {
    public:
      SLSCPreconditionerFactory(
        RCP<Teuchos::ParameterList>   F_SolveStrategy,
        RCP<Teuchos::ParameterList>   H_SolveStrategy,
        bool uniformMesh=true
        );
      RCP<Thyra::PreconditionerBase<double> > createPrec() const;
      void initializePrec(const RCP<const LinearOpSourceBase<double> >
                          &fwdOpSrc,
                          Thyra::PreconditionerBase<double> *precOp,
                          const ESupportSolveUse
                          supportSolveUse = SUPPORT_SOLVE_UNSPECIFIED) const;

      void setGammaParameters(Teuchos::RCP<Teuchos::ParameterList> paramList);
      void setAlphaParameters(Teuchos::RCP<Teuchos::ParameterList> paramList);

[...]
```

The class *SLSCOperatorSource* saves all necessary data to compute the preconditioning operator. The constructor of *SLSCOperatorSource* expects the four block matrices of (1.4) and the diagonal of the mass matrix as arguments.

```
      SLSCOperatorSource(Epetra_CrsMatrix* S00,
                         Epetra_CrsMatrix* S01,
                         Epetra_CrsMatrix* S10,
                         Epetra_CrsMatrix* S11,
                         Epetra_Vector* Qu);
```

The resulting object can then be passed to the initialization function of the *SLSCPreconditionerFactory*.

The class *SLSCPreconditionerFactory* is an actual factory. Its product, a *Thyra::PreconditionerBase* object, finally contains the preconditioning operator and can be used within the Thyra package to define preconditioned solvers. The constructor of *SLSCPreconditionerFactory* takes three arguments:

```
     SLSCPreconditionerFactory(
        RCP<Teuchos::ParameterList>   F_SolveStrategy,
        RCP<Teuchos::ParameterList>   H_SolveStrategy,
        bool uniformMesh=true
        );
```

F_SolveStrategy and *H_SolveStrategy* define the parameters for solving F^{-1} and for H^{-1}, respectively. H is set to $(BQ^{-1}B^T + \gamma C)$ if $uniformMesh = true$ and to $(BQ^{-1}B^T + \gamma \tilde{D}_r^{\frac{1}{2}} C \tilde{D}_r^{\frac{1}{2}})$ if $uniformMesh = false$ (compare with (3.10) and (3.12)). To solve F^{-1} and H^{-1} Stratimikos is used. Therefore the values in *F_SolveStrategy* and *H_SolveStrategy* have to be chosen appropriate for Stratimikos (for a description of valid values see [4]). The boolean

uniformMesh indicates if the mesh used to create the approximation matrices was uniform or not. The setting *uniform* = *false* works for uniform meshes too, with the only disadvantage of a slightly higher computational effort. Note that if *uniform* = *false* is chosen, the approximation may not be stable, since C would be zero and the algorithm would divide by zero while computing D_r.

To compute α and γ two eigenvalue problems have to be solved. If desired, their solver parameters can be set with:

```
1        void setGammaParameters(Teuchos::RCP<Teuchos::ParameterList> paramList);
2        void setAlphaParameters(Teuchos::RCP<Teuchos::ParameterList> paramList);
```

The Anasazi package is used to solve the eigenvalue problems and therefore the values in *paramList* have to be appropriate for Anasazi in both cases (for a description of valid values see [1]). If the parameters are not set manually default values will be used. These are:

```
1  // The eigenvalue of interest: Largest Magnitude. Don't change that!
2  RCP<Anasazi::SortManager<ScalarType,MV,OP> > MySort =
3    rcp( new Anasazi::BasicSort<ScalarType,MV,OP>( "LM" );
4  paramList->set( "Sort Manager", MySort );
5  // block size to be used by the underlying block Davidson solver
6  paramList->set( "Block Size", 1 );
7  // number of blocks allocated for the Krylov basis
8  paramList->set( "Num Blocks", 3 );
9  // maximum number of restarts the underlying solver is allowed to perform
10 paramList->set( "Maximum Restarts", 500 );
11 paramList->set( "Convergence Tolerance", 1.0e-2 );
```

Tests have shown that a tolerance of $1.0e-2$ is an appropriate choice to achieve a good preconditioner while minimizing the computational costs.

After declaring the solving settings, the preconditioner operator of type *Thyra::PreconditionerBase* can be created. At first, a (uninitialized) preconditioner object is generated with

```
1        RCP<Thyra::PreconditionerBase<double> > createPrec() const;
```

which then is passed to the initialization routine:

```
1        void initializePrec(const RCP<const LinearOpSourceBase<double> >
2                            &fwdOpSrc,
3                            Thyra::PreconditionerBase<double> *precOp,
4                            const ESupportSolveUse
5                            supportSolveUse = SUPPORT_SOLVE_UNSPECIFIED) const;
```

initializePrec is the main function of *SLSCPreconditionerFactory*. With the data given in *fwdOpSr* the precondition operator is computed as described in section 3.2 with the user-defined solver parameters. The result is saved in *precOp* and can then be used like any other Thyra preconditioner object. The *supportSolveUse* is not evaluated by *initializePrec* and can be ignored.

In conclusion, the typical use of the SLSC preconditioner would be:

```
1        // F, Bt, B, C are of type Epetra::Epetra_CrsMatrix matrices, Qu of type Epetra::Vector
2        RCP<const LinearOpSourceBase<double> > mySLSCopSrcRcp = rcp(new SLSCOperatorSource(F,Bt,
             B,C,Qu));
3        // invFParams, invHParams are of type (Teuchos::RCP<Teuchos::ParameterList> paramList,
             uniform of type bool
4        RCP<PreconditionerFactoryBase<double> > merosPrecFac  = rcp(new
             SLSCPreconditionerFactory(invFParams, invHParams, uniform));
5        RCP<PreconditionerBase<double> > Prcp   = merosPrecFac->createPrec();
6        merosPrecFac->initializePrec(mySLSCopSrcRcp, &*Prcp);
```

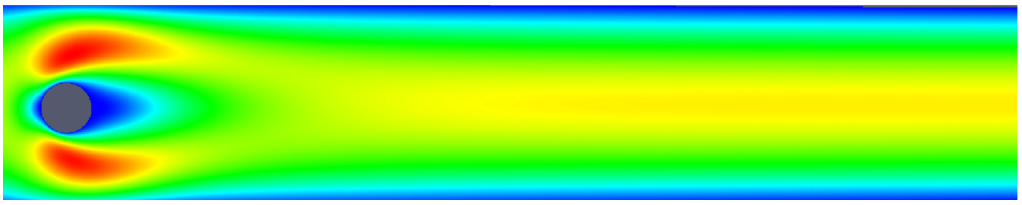


Figure 1: The stationary ($\alpha = 0$) solution of the example problem. The picture shows the velocity speed in x-direction. ν is set $0,01$.

The resulting *Prcp* of type *PreconditionerBase* can then be passed to a Thyra solver as preconditioner. A complete example is given in the next section.

4.3 Example

This section presents an example use-case of the SLSC preconditioner which will be later used for the numerical experiments as well. The considered domain is a rectangle with a circle placed in the middle. The boundary condition on the left wall is set to zero speed in y-direction and to an quadratic function in the x-direction with 0 in the top and bottom corner and its maximum in the middle of the wall. The top and bottom borders as well as the inner circle satisfy the "no-slip" condition, that is the fluid sticks at the borders. An typical solution is shown in Figure 1.

To simplify matters, just the stationary case is shown here. Let $\mathbf{H}^1_E$ and $\mathbf{H}^1_{E_0}$ be suitable function spaces for the unknown velocity function and for test functions, respectively. The weak formulation of the Navier Stokes equation (1.1), (1.2) can then be written as: Find $\vec{u} \in \mathbf{H}^1_E$ and $p \in L_2(\Omega)$ such that

$$\begin{aligned} \nu \int_\Omega \nabla\vec{u} : \nabla\vec{v} + \int_\Omega (\vec{u}\cdot\nabla\vec{u})\cdot\vec{v} - \int_\Omega p(\nabla\cdot\vec{v}) &= \int_\Omega \vec{f}\cdot\vec{v} \\ \int_\Omega q(\nabla\cdot\vec{u}) &= 0 \end{aligned} \tag{4.1}$$

$\forall q \in L_2(\Omega)$ and $\forall \vec{v} \in \mathbf{H}^1_{E_0}$.

The first equation is nonlinear and therefore needs to be solved by a nonlinear solver. A common choice is Picard iteration, which starts with an 'initial value' (u_0, p_0) and computes a sequence of iterates $(\vec{u}_1, p_1)$, $(\vec{u}_2, p_2)$, $(\vec{u}_3, p_3)$, ... If the initial value lies in the ball of convergence, the iterates will converge to the solution of (4.1). Each Picard iterate is computed by a fixed point strategy where the nonlinear term is evaluated in the current

velocity value: Find $\vec{u}_{k+1} \in \mathbf{H}^1_E$ and $p_{k+1} \in L_2(\Omega)$ such that

$$\begin{aligned} \nu \int_\Omega \nabla \vec{u}_{k+1} : \nabla \vec{v} + \int_\Omega (\vec{u}_k \cdot \nabla \vec{u}_{k+1}) \cdot \vec{v} - \int_\Omega p_{k+1} (\nabla \cdot \vec{v}) &= \int_\Omega \vec{f} \cdot \vec{v} \\ \int_\Omega q (\nabla \cdot \vec{u}_{k+1}) &= 0 \end{aligned} \tag{4.2}$$

$\forall q \in L_2(\Omega)$ and $\forall \vec{v} \in H^1_{E_0}$. Equation (4.2) is also known as *Oseen system*.

Using stable finite elements, it would be possible to directly discretize (4.2). A linear system with 0 in the (2,2) block would be the result and could be preconditioned by LSC.

In this example we will rather assume the usage of unstable elements $\mathbf{P}_1 - \mathbf{P}_1$. To stabilize (4.2), a method described in [11], p. 243 is used: The compressibility constraint in (4.2) is perturbed by a Laplacian term so that the weak form becomes

$$\begin{aligned} \nu \int_\Omega \nabla \vec{u}_{k+1} : \nabla \vec{v} + \int_\Omega (\vec{u}_k \cdot \nabla \vec{u}_{k+1}) \cdot \vec{v} - \int_\Omega p_{k+1} (\nabla \cdot \vec{v}) &= \int_\Omega \vec{f} \cdot \vec{v} \\ \int_\Omega q (\nabla \cdot \vec{u}_{k+1}) + \beta h^2 \int_\Omega \nabla q \cdot \nabla p_{k+1} &= 0 \end{aligned} \tag{4.3}$$

The parameter β in the stabilization term is user specified and h is the element size for uniform meshes. Although this stabilization method is primarily not designed for non-uniform meshes, in this example it is used anyway. Then h is the size of the element which is evaluated during the assembling process in the finite element method.

Applying the finite element method to (4.3) results to a linear equation

$$\begin{bmatrix} F_k & B^T \\ B & -C \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta p_k \end{bmatrix} = r_k \tag{4.4}$$

where $\Delta u_k := u_{k+1} - u_k$ and $\Delta p_k := p_{k+1} - p_k$.

In the following it will be demonstrated how to solve (4.4) with the SLSC preconditioner by going step by step through an example code. First the matrix data is loaded:

```
1        // We need a velocity space map and a pressure space map.
2        const Epetra_Map* velocityMap = new Epetra_Map(19706, 0, Comm); // Dimension of velocity
             space
3        const Epetra_Map* pressureMap = new Epetra_Map(9853, 0, Comm); // Dimension of pressure
             space
4        bool  uniform=false;
5
6        // Read matrix and vector blocks into Epetra_Crs matrices
7        Epetra_CrsMatrix* FMatrix(0);
8        MatrixMarketFileToCrsMatrix(filenameF,
9                                    *velocityMap,
10                                   *velocityMap, *velocityMap,
11                                   FMatrix);
12
13
```

```
14        [...]
15
16
17        // Wrap Epetra operators into Thyra operators.
18        RCP<LinearOpBase<double> >
19          tmpF = nonconstEpetraLinearOp(rcp(FMatrix,false));
20        const LinearOperator<double> F = tmpF;
21
22
23        [...]
```

The variable *uniform* indicates that the used mesh was non-uniform and will be passed to the SLSC preconditioner. Line 7-11 and 18-20 are repeated to load Bt, B, C and the right hand side vectors *rhs_vel* and *rhs_press*.

Next, the saddle point system *blockOp* and the solution vector are created:

```
1       // Build the block saddle operator with F, Bt, B, and -C.
2       LinearOperator<double> blockOp = block2x2(F, Bt, B, -1.0*C);
3
4        // Get the domain and range product spaces from the block operator.
5        VectorSpace<double> domain = blockOp.domain();
6        VectorSpace<double> range = blockOp.range();
7
8        // Build a solution vector and initialize it to zero.
9        Vector<double> solnblockvec = domain.createMember();
10       zeroOut(solnblockvec);
```

With this preparation the SLSC preconditioner can be build. This is done in 3 steps:

```
1        // 1) Load the ParameterList for inv(F) solve and for inv(H) solve
2        ParameterXMLFileReader reader("./aztecFParam.xml");
3        RCP<ParameterList> aztecFParams = rcp(new ParameterList(reader.getParameters()));
4
5        ParameterXMLFileReader readerH("./aztecHParam.xml");
6        RCP<ParameterList> aztecHParams = rcp(new ParameterList(readerH.getParameters()));
7
8        // 2) Make an SLSCOperatorSource that contains  blockOp
9        if (DEBUG>0) std::cout  << "Create SLSCOperatorSource...";
10       RCP<const LinearOpSourceBase<double> > mySLSCopSrcRcp = rcp(new SLSCOperatorSource(
             FMatrix,BtMatrix,BMatrix,CMatrix,QuVector));
11
12
13       // 3) Build the SLSC block preconditioner factory.
14       if (DEBUG>0) std::cout << "Create preconditioner..." << endl;
15       RCP<PreconditionerFactoryBase<double> > merosPrecFac  = rcp(new
             SLSCPreconditionerFactory(aztecFParams, aztecHParams, uniform));
16       RCP<PreconditionerBase<double> > Prcp   = merosPrecFac->createPrec();
17       merosPrecFac->initializePrec(mySLSCopSrcRcp, &*Prcp);
```

Step 1 sets the parameters for the preconditioner. At least two parameters are necessary, namely for solving the inverse of F and of H. Step 2 creates the source object for the preconditioner. It contains all necessary data for the computation of the preconditioned matrix. Step 3 builds the preconditioner itself. Therefore, first a preconditioner factory is created which is then be used to build the preconditioner object $Prcp$. No parameter for the eigenvalue problem in the *PreconditionerFactoryBase* object are defined, so the default parameters will be used. See section (4.2) for more details.

After creating the SLSC preconditioner it is time to set up the preconditioned inverse object of *blockOp*. In this example, *blockOp* will be solved with $GMRESR$ which is provided by the Aztec[2] package:

[2]Unfortunately, GMRESR is not activated by default. See the installation section. More about GMRESR can be found in the numerical results section

```
1       // Set up parameter list and AztecOO solver
2        RCP<ParameterList> aztecSaddleParams  = rcp(new ParameterList("
             aztecOOSaddleSolverFactory"));
3        // forward solve settings
4        aztecSaddleParams->sublist("Forward Solve").set("Max Iterations", 500);
5        aztecSaddleParams->sublist("Forward Solve").set("Tolerance", 1.0e-6);
6        // aztecOO solver settings
7        aztecSaddleParams->sublist("Forward Solve")
8          .sublist("AztecOO Settings").set("Aztec Solver", "GMRESR");
9        aztecSaddleParams->sublist("Forward Solve")
10         .sublist("AztecOO Settings").set("Aztec Preconditioner", "none");
11       aztecSaddleParams->sublist("Forward Solve")
12         .sublist("AztecOO Settings").set("Size of Krylov Subspace", 20);
```

Note that no preconditioner is defined here, since we want to use the externally created SLSC preconditioner. This parameterlist is now passed to a $LinearOpWithSolveFactoryBase$ object,

```
1       RCP<LinearOpWithSolveFactoryBase<double> >   aztecSaddleLowsFactory = rcp(new
            AztecOOLinearOpWithSolveFactory());
2        aztecSaddleLowsFactory->setParameterList(aztecSaddleParams);
```

which produces a $LinearOpWithSolveBase$. With this $LinearOpWithSolveBase$ finally the inverse of *blockOp* can be created:

```
1      // Set up the preconditioned inverse object and do the solve!
2      RCP<LinearOpWithSolveBase<double> > rcpAztecSaddle   = aztecSaddleLowsFactory->createOp();
3      initializePreconditionedOp<double>(*aztecSaddleLowsFactory,
4                                          blockOp.ptr(),
5                                          Prcp,
6                                         &*rcpAztecSaddle );
7      RCP<LinearOpBase<double> > tmpSaddleInv = rcp(new DefaultInverseLinearOp<double>(
           rcpAztecSaddle));
8
9      LinearOperator<double> saddleInv = tmpSaddleInv;
```

Using the $*$ operator on the *saddleInv* operator with a vector will now automatically solve the linear system with GMRESR and SLSC preconditioning. Thus, all left to do is

```
1     // Do the solve!
2        solnblockvec = saddleInv * rhs;
```

Finally, the residuum is computed and checked if it is smaller than the requested tolerance:

```
1        // Check our results.
2        Vector<double> residvec = blockOp * solnblockvec - rhs;
3        double normResvec = norm2(residvec);
4
5        double saddleTol=1.0e-6;  // Residuum tolerance
6        if(normResvec < 10.0*saddleTol)
7          {
8            cerr << "Example PASSED!" << endl;
9            return 0;
10         }
11       else
12         {
13           cerr << "Example FAILED!" << endl;
14   return 1;
15         }
```

5 Numerical results

The numerical results presented here are based on the example discussed in section 4.3. The tests have been performed on a HP nc8430 laptop with a

2,0GHz Intel Core 2 Dual processor (only one processor unit was used for computation though) and 3GB of memory.

To subdivide the domain (see figure 1) the mesh generation software Triangle [7] is used. All following results have been calculated with a uniform mesh unless otherwise stated. The stabilized weak formulation (4.3) is discretized with $\mathbf{P}_1 - \mathbf{P}_1$ elements. As FEM package Sundance [5] is used which is included in Trilinos as well. The user-defined stabilization parameter β in (4.3) is chosen in such a way as to achieve a stable solution with minimal computational costs. More details about the effects on β on the solution are given later. The resulting linear problems have the form

$$\begin{bmatrix} F_k & B^T \\ B & -C \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta p_k \end{bmatrix} = r_k$$

and are the input for the following benchmarks.

Consider table 2 and 3. The problems in table 2 are generated from the stationary Navier Stokes equation. By comparison, table 3 shows the results for the same problems but for instationary flow (that is $\alpha = 1$). In both cases, the 4. Picard iteration (of the first timestep) is used as benchmark.

Four different solver techniques are applied to these linear systems (the term (ILU) after a solver name indicates preconditioning with incomplete LU factorization):

- **Direct**: LU decomposition with MUMPS [3], a fast iterative solver for large sparse linear systems.
- **SLSC direct**: GMRESR with SLSC preconditioning. F^{-1} is solved with SuperLU [6] (ILU) and H^{-1} with DSCPACK [2] (ILU), two direct solvers for large sparse linear systems.
- **SLSC iterative**: GMRESR with SLSC preconditioning. F^{-1} and H^{-1} are solved iteratively with GMRES (ILU) and CG (ILU), respectively. The accuracy of these subsolvers is set to $1.0e-3$.
- **SLSC mixed**: GMRESR with SLSC preconditioning. F^{-1} is solved iteratively with GMRES (ILU) while for H^{-1} DSCPACK (ILU) is applied. The accuracy of GMRES is set to $1.0e-3$. This method is just used for the instationary Navier stokes equation as it brings significant results in this case.

GMRESR is a variant of the generalized minimal residual method published by Vorst und Vuik [14]. By nesting the GMRES method it allows the use of a different preconditioner at each iteration. This is necessary because

Solver	**Dimension of the problem**	31.992	231.705	459.984	605.058
Direct	**Time** (sec)	1,3	17	44	-
	Mermoy usage	60	500	1000	-
SLSC direct	**Time** (sec)	4,3	45	103	-
	Memory usage (MB)	60	500	1000	-
	GMRESR iterations	51	58	58	-
SLSC iterative	**Time** (sec)	15	559	1736	-
	Memory usage (MB)	60	400	750	-
	GMRESR iterations	51	58	58	-

Table 2: Comparison of different solvers for the stationary Navier Stokes equation. The largest problem could not be solved due to a small system memory. The problem parameters are $\alpha = 0$, $\nu = 0,01$, $\beta = 1.0$ and $k = 4$.

the preconditioning operator is solved with iterative methods in "SLSC iterative" and "SLSC mixed". In this case, the provided solution is only an approximation of the exact solution what can also be seen as a varying of the preconditioner. This property even allows to reduce the accuracy for the solution of the preconditioning operator which ends up in an increasing solving speed. On the other hand, the loss of accuracy will cause slightly different iteration counts of the GMRESR between the different SLSC solvers.

The initial iterate of GMRESR is set to zero and the required tolerance to $1.0e-6$. That is, the last iterate $[\Delta u_k^i, \Delta p_k^i]$ must satisfy

$$|| \begin{bmatrix} F_k & B^T \\ B & -C \end{bmatrix} \begin{bmatrix} \Delta u_k^i \\ \Delta p_k^i \end{bmatrix} - r_k|| \leq 10^{-6}$$

Three main points can be observed from table 2 and 3. Firstly, the iteration count needed with SLSC preconditioning are almost independent from the problem size. Therefore SLSC satisfies an important property of a reasonable preconditioner.

Secondly, the memory usage used by SLSC methods is usually less than that of "Direct". The reason is that no LU decomposition has to be saved during the solving process. Instead, in order to compute the next iteration vector, GMRESR needs just the last iteration vector and a set of base vectors. With a special restarting technique the maximum necessary memory of GMRESR can be even limited to an upper bound. Though, if the subproblems of the SLSC are solved directly, then LU decompositions of F and H are created and have to be saved. This can be recognized in an almost equal memory usage of "Direct" and "SLSC direct".

Thirdly, the benchmarks show that the SLSC methods are slower than

Solver	**Dimension of the problem**	31.992	231.705	459.984	605.058
Direct	**Time** (sec)	1,3	16	47	68
	Mermoy usage (MB)	60	600	1000	1400
SLSC direct	**Time** (sec)	3,6	29	66	93
	Memory usage (MB)	80	500	1000	1300
	GMRESR iterations	34	25	23	23
SLSC iterative	**Time** (sec)	6,3	58	139	196
	Memory usage	50	250	650	750
	GMRESR iterations	36	28	25	26
SLSC mixed	**Time** (sec)	3,4	27	59	82
	Memory usage (MB)	50	350	700	900
	GMRESR iterations	36	28	25	26

Table 3: Comparison of different solvers for the instationary Navier Stokes equation. The solved linear systems arise from the 4. Picard iteration during the first timestep. The problem parameters are $\alpha = 1$, $\nu = 0,01$, $\beta = 1.0$ and $k = 4$.

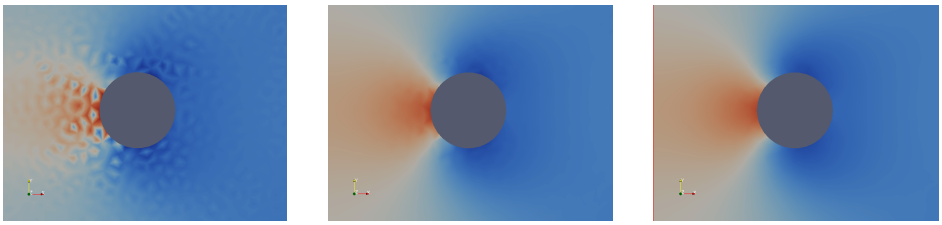
"Direct" for the testcases. In the stationary case "SLSC direct" is the fastest SLSC method but still more than twice as slow as "Direct". The high running times for "SLSC iterative" is caused by large iteration counts of the methods used for the subproblems. Here, more optimization of the preconditioning subsolvers is necessary to achieve better results.

In the instationary case the fastest SLSC method is "SLSC mixed" where an iterative solver is used for F^{-1} and a direct solver for H^{-1}. This combines the fast running times of the direct solver with the memory saving calculation of the iterative method. The result is an algorithm which has almost the same running time than that of "Direct" but saves about 30% system memory during the calculation of the larger test problems. On the other hand, the solving times are significantly higher if iterative methods are used to solve both subproblems arising in the SLSC preconditioner ("SLSC iterative"). The reason is following: If the iteration count is limited, the complexity of GMRES and CG methods is $O(n)$, but with a high constants [13]. However, a LU decomposition needs $O(n^3)$ operations, with a relatively small constant. Thus, iterative methods become interesting not before the problem size exceeds a certain size. These dimension can be reached for example if the problem is to be computed on a 3D domain.

Table 4 shows the relation of the stabilization parameter β and the iteration count of GMRESR. Figure 2 presents the associated solutions for three values of β. As can be seen, a smaller choice of β leads to fewer iterations

Beta	0,01	0,1	0,5	1.0	5.0
Iterationen	21	22	38	51	112

Table 4: Correlation of the iteration count of "SLSC direct" and the stabilization parameter. The problem dimension is 31.992, the problem parameters are $\alpha = 0$ and $\nu = 0,01$.



(a) $\beta = 0,01$ (b) $\beta = 1,0$ (c) $\beta = 5,0$

Figure 2: The pressure solutions of table 4 zoomed in to the inner circle. The color represents the pressure value.

but also to less stabilization. If β is chosen too small, the system is not sufficiently stabilized and the solution in the pressure space contains alternating jumps. This instability effect is also known as *checkerboard mode*.

In table 5 the correlation of the viscosity parameter ν and the iteration count of GMRESR is shown. The choice of β is made by minimizing the iterations but still obtaining stable solutions.

Finally, table 6 compares the iterations of GMRESR for uniform and non-uniform meshes.

ν	Iterations	Time (sec)	β
1,0	>500	>30	0,05
0,1	61	4,9	0,05
0,01	51	4,3	1,0
0,001	34	3,5	1,0
0,0001	No convergence of outer Picard iteration		

Table 5: Influence of the parameter ν on the iteration count of "'SLSC direct"'. The 4. Picard iteration of the stationary Navier Stokes equation is used. The problem dimension is 31992.

Mesh	Dimension	Time (sec)	Iterations
Uniform	6390	0,77	39
Non-uniform	5985	0,76	51

Table 6: Comparison of "SLSC direct" on a uniform and a non-uniform mesh. The problem parameters are $\alpha = 0$, $\beta = 0,5$ and $\nu = 0,01$.

6 Outlook

As shown in the benchmarks, GMRESR with SLSC preconditioning is a competitive method to solve the Navier Stokes equations, especially for the instationary case and fine meshes. The higher the problem dimension becomes, the better is the running time of the SLSC methods compared to a direct solve. Therefore it would be interesting to see how this solver works for even higher dimensions as they arise on 3D domains.

In addition, the numerical tests demonstrate that the choice of the subsolvers for the SLSC preconditioner play a decisive role for the total solving speed. Methods like algebraic multigrid are available in Trilinos and could be used to improve the running time of the SLSC methods.

Finally, due to the design of Trilinos the presented algorithm should run on parallel machines as well, but what was never tested within this work.

7 Installation

To add the SLSC preconditioner into the Meros package, all files in the "Meros" folder on the project DVD have to be copied into the "packages/meros" folder of the Trilinos code. Then install Trilinos as described in the documentation. In order to compile Meros with SLSC successfully following packages in Trilinos must be activated:

- Thyra
- Epetra
- Stratimikos
- Anasazi

Depending on the solver which should be used for the occurring subproblems additional packages are needed, e.g. Amesos for direct solvers and Aztecoo for Krylov subspace methods.

The preconditioned linear system is usually solved with GMRESR. Unfortunately, in Trilinos 8.0 the GMRESR is not activated in Stratimikos. In order to enable it open *stratimikos/adapters/aztecoo/src/AztecOOParameterList.cpp* and exchange Line 192-193

```
1       tuple<std::string>("CG","GMRES","CGS","TFQMR","BiCGStab","LU"),
2       tuple<int>(AZ_cg,AZ_gmres,AZ_cgs,AZ_tfqmr,AZ_bicgstab,AZ_lu),
```

to

```
1       tuple<std::string>("CG","GMRES","GMRESR","CGS","TFQMR","BiCGStab","LU"),
2       tuple<int>(AZ_cg,AZ_gmres,AZ_GMRESR,AZ_cgs,AZ_tfqmr,AZ_bicgstab,AZ_lu),
```

Besides the preconditioner code, the project DVD includes all benchmark data from this paper. The "Benchmarks" folder contains the used meshes, the parameter files and all problem matrices which where solved in section 5. With these, it is easy to reproduce the benchmark results and extend the tests. As well, the example program described in section 4.3 can be found in the *meros/examples/saddle_slsc* folder. Here a brief overview of the files in this folder:

- *saddle_direkt.cpp* - Solves a test matrix directly. The parameters are set in $stratimikosDirektParam.xml$.
- *saddle_slsc.cpp* - Solves a test matrix with GMRESR and SLSC. The parameters are set in $stratimikosFParam.xml$ and $stratimikosHParam.xml$.
- *Sundance_Test/NavStokG2PicSolvergetBlockMatricesv4c.cpp* - Solves the stationary Navier Stokes equation with a given mesh and saves the arising matrices. Can be used to create new test matrices.
- *Sundance_Test_instat/NavStokG2PicSolvergetBlockMatricesv4c.cpp* - Solves the instationary Navier Stokes equation with a given mesh and saves the arising matrices. Can be used to create new test matrices.
- *matlab/createQuvec*.m* - Matalab tool to extract the diagonal elements of the mass matrix used by the SLSC preconditioner.

References

[1] Anasazi: A block eigensolvers package. `http://trilinos.sandia.gov/packages/docs/r8.0/packages/anasazi/doc/html/index.html`.

[2] Dscpack: Domain-separator codes for solving sparse linear systems. `http://www.cse.psu.edu/~raghavan/Dscpack/`.

[3] Mumps: A parallel sparse direct solver. `http://graal.ens-lyon.fr/MUMPS/`.

[4] Stratimikos: Thyra-based strategies for linear solvers. `http://trilinos.sandia.gov/packages/docs/r8.0/packages/stratimikos/doc/html/index.html`.

[5] Sundance: A system for rapid development of high-performance parallel finite-element solutions of partial differential equations. `http://www.math.ttu.edu/~klong/Sundance/html`.

[6] Superlu: A general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations. `http://crd.lbl.gov/~xiaoye/SuperLU/`.

[7] Triangle: A two-dimensional quality mesh generator and delaunay triangulator. `http://www.cs.cmu.edu/~quake/triangle.html`.

[8] Howard Elman, V. E. Howle, John Shadid, Robert Shuttleworth, and Ray Tuminaro. A taxonomy and comparison of parallel block multilevel preconditioners for the incompressible navier-stokes equations. *J. Comput. Phys.*, 227(3):1790–1808, 2008.

[9] Howard Elman, Victoria E. Howle, John Shadid, Robert Shuttleworth, and Ray Tuminaro. Block preconditioners based on approximate commutators. *SIAM Journal on Scientific Computing*, 27(5):1651–1668, 2006.

[10] Howard Elman, Victoria E. Howle, John Shadid, David Silvester, and Ray Tuminaro. Least squares preconditioners for stabilized discretizations of the navier-stokes equations. *SIAM Journal on Scientific Computing*, 30(1):290–311, 2007.

[11] Andy Wathen Howard Elman, David Silvester. *Finite Elements and Fast Iterative Solver with applications in incompressible fluid dynamics.* Oxford Science Publications, 2005.

[12] Malcolm F. Murphy, Gene H. Golub, and Andrew J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM Journal on Scientific Computing*, 21(6):1969–1972, 2000.

[13] Burlisch Stoer. *Numerische Mathematik II.* Springer Verlag Berlin, 2000.

[14] C. Vuik, Henk A., and Van Der Vorst HA. GMRESR: A family of nested GMRES methods. Technical report, 1994.